# Students' Guide to Raft

Posted on Mar 16, 2016 — shared on Hacker News, Twitter, and Lobsters

For the past few months, I have been a Teaching Assistant for MIT's 6.824 Distributed Systems class. The class has traditionally had a number of labs building on the Paxos consensus algorithm, but this year, we decided to make the move to Raft. Raft was "designed to be easy to understand", and our hope was that the change might make the students' lives easier.

This post, and the accompanying Instructors' Guide to Raft post, chronicles our journey with Raft, and will hopefully be useful to implementers of the Raft protocol and students trying to get a better understanding of Raft's internals. If you are looking for a Paxos vs Raft comparison, or for a more pedagogical analysis of Raft, you should go read the Instructors' Guide. The bottom of this post contains a list of questions commonly asked by 6.824 students, as well as answers to those questions. If you run into an issue that is not listed in the main content of this post, check out the Q&A. The post is quite long, but all the points it makes are real problems that a lot of 6.824 students (and TAs) ran into. It is a worthwhile read.

## Background

Before we dive into Raft, some context may be useful. 6.824 used to have a set of Paxos-based labs that were built in Go; Go was chosen both because it is easy to learn for students, and because is pretty well-suited for writing concurrent, distributed applications (goroutines come in particularly handy). Over the course of four labs, students build a fault-tolerant, sharded key-value store. The first lab had them build a consensus-based log library, the second added a key value store on top of that, and the third sharded the key space among multiple fault-tolerant clusters, with a fault-tolerant shard master handling configuration changes. We also had a fourth lab in which the students had to handle the failure and recovery of machines, both with and without their disks intact. This lab was available as a default final project for students.

This year, we decided to rewrite all these labs using Raft. The first three labs were all the same, but the fourth lab was dropped as persistence and failure recovery is already built into Raft. This article will mainly discuss our experiences with the first lab, as it is the one most directly related to Raft, though I will also touch on building applications on top of Raft (as in the second lab).

Raft, for those of you who are just getting to know it, is best described by the text on the protocol's web site:

> Raft is a consensus algorithm that is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems. We hope Raft will make consensus

> available to a wider audience, and that this wider audience will be able to develop a variety of higher quality consensus-based systems than are available today.

Visualizations like this one give a good overview of the principal components of the protocol, and the paper gives good intuition for why the various pieces are needed. If you haven't already read the extended Raft paper, you should go read that before continuing this article, as I will assume a decent familiarity with Raft.

As with all distributed consensus protocols, the devil is very much in the details. In the steady state where there are no failures, Raft's behavior is easy to understand, and can be explained in an intuitive manner. For example, it is simple to see from the visualizations that, assuming no failures, a leader will eventually be elected, and eventually all operations sent to the leader will be applied by the followers in the right order. However, when delayed messages, network partitions, and failed servers are introduced, each and every if, but, and and, become crucial. In particular, there are a number of bugs that we see repeated over and over again, simply due to misunderstandings or oversights when reading the paper. This problem is not unique to Raft, and is one that comes up in all complex distributed systems that provide correctness.

## Implementing Raft

The ultimate guide to Raft is in Figure 2 of the Raft paper. This figure specifies the behavior of every RPC exchanged between Raft servers, gives various invariants that servers must maintain, and specifies when certain actions should occur. We will be talking about Figure 2 a lot in the rest of this article. It needs to be followed to the letter.

Figure 2 defines what every server should do, in ever state, for every incoming RPC, as well as when certain other things should happen (such as when it is safe to apply an entry in the log). At first, you might be tempted to treat Figure 2 as sort of an informal guide; you read it once, and then start coding up an implementation that follows roughly what it says to do. Doing this, you will quickly get up and running with a mostly working Raft implementation. And then the problems start.

In fact, Figure 2 is extremely precise, and every single statement it makes should be treated, in specification terms, as **MUST**, not as **SHOULD**. For example, you might reasonably reset a peer's election timer whenever you receive an `AppendEntries` or `RequestVote` RPC, as both indicate that some other peer either thinks it's the leader, or is trying to become the leader. Intuitively, this means that we shouldn't be interfering. However, if you read Figure 2 carefully, it says:

> If election timeout elapses without receiving `AppendEntries` RPC from current leader or granting vote to candidate: convert to candidate.

The distinction turns out to matter a lot, as the former implementation can result in significantly reduced liveness in certain situations.

**The importance of details**

To make the discussion more concrete, let us consider an example that tripped up a number of 6.824 students. The Raft paper mentions heartbeat RPCs in a number of places. Specifically, a leader will occasionally (at least once per heartbeat interval) send out an `AppendEntries` RPC to all peers to prevent them from starting a new election. If the leader has no new entries to send to a particular peer, the `AppendEntries` RPC contains no entries, and is considered a heartbeat.

Many of our students assumed that heartbeats were somehow "special"; that when a peer receives a heartbeat, it should treat it differently from a non-heartbeat `AppendEntries` RPC. In particular, many would simply reset their election timer when they received a heartbeat, and then return success, without performing any of the checks specified in Figure 2. This is extremely dangerous. By accepting the RPC, the follower is implicitly telling the leader that their log matches the leader's log up to and including the `prevLogIndex` included in the `AppendEntries` arguments. Upon receiving the reply, the leader might then decide (incorrectly) that some entry has been replicated to a majority of servers, and start committing it.

Another issue many had (often immediately after fixing the issue above), was that, upon receiving a heartbeat, they would truncate the follower's log following `prevLogIndex`, and then append any entries included in the `AppendEntries` arguments. This is also not correct. We can once again turn to Figure 2:

> If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it.

The if here is crucial. If the follower has all the entries the leader sent, the follower **MUST NOT** truncate its log. Any elements following the entries sent by the leader **MUST** be kept. This is because we could be receiving an outdated `AppendEntries` RPC from the leader, and truncating the log would mean "taking back" entries that we may have already told the leader that we have in our log.

## Debugging Raft

Inevitably, the first iteration of your Raft implementation will be buggy. So will the second. And third. And fourth. In general, each one will be less buggy than the previous one, and, from experience, most of your bugs will be a result of not faithfully following Figure 2.

When debugging, Raft, there are generally four main sources of bugs: livelocks, incorrect or incomplete RPC handlers, failure to follow The Rules, and term confusion. Deadlocks are also a common problem, but they can generally be debugged by logging all your locks and unlocks, and figuring out which locks you are taking, but not releasing. Let us consider each of these in turn:

## Livelocks

When your system livelocks, every node in your system is doing something, but collectively your nodes are in such a state that no progress is being made. This can happen fairly easily in Raft, especially if you do not follow Figure 2 religiously. One livelock scenario comes up especially often; no leader is being elected, or once a leader is elected, some other node starts an election, forcing the recently elected leader to abdicate immediately.

There are many reasons why this scenario may come up, but there is a handful of mistakes that we have seen numerous students make:

- Make sure you reset your election timer exactly when Figure 2 says you should. Specifically, you should only restart your election timer if a) you get an `AppendEntries` RPC from the current leader (i.e., if the term in the `AppendEntries` arguments is outdated, you should not reset your timer); b) you are starting an election; or c) you grant a vote to another peer.

  This last case is especially important in unreliable networks where it is likely that followers have different logs; in those situations, you will often end up with only a small number of servers that a majority of servers are willing to vote for. If you reset the election timer whenever someone asks you to vote for them, this makes it equally likely for a server with an outdated log to step forward as for a server with a longer log.

  In fact, because there are so few servers with sufficiently up-to-date logs, those servers are quite unlikely to be able to hold an election in sufficient peace to be elected. If you follow the rule from Figure 2, the servers with the more up-to-date logs won't be interrupted by outdated servers' elections, and so are more likely to complete the election and become the leader.

- Follow Figure 2's directions as to when you should start an election. In particular, note that if you are a candidate (i.e., you are currently running an election), but the election timer fires, you should start another election. This is important to avoid the system stalling due to delayed or dropped RPCs.

- Ensure that you follow the second rule in "Rules for Servers" before handling an incoming RPC. The second rule states:

> If RPC request or response contains term `T > currentTerm`: set `currentTerm = T`, convert to follower (§5.1)

For example, if you have already voted in the current term, and an incoming `RequestVote` RPC has a higher term that you, you should first step down and adopt their term (thereby resetting `votedFor`), and then handle the RPC, which will result in you granting the vote!

**Incorrect RPC handlers**

Even though Figure 2 spells out exactly what each RPC handler should do, some subtleties are still easy to miss. Here are a handful that we kept seeing over and over again, and that you should keep an eye out for in your implementation:

- If a step says "reply false", this means you should reply immediately, and not perform any of the subsequent steps.
- If you get an `AppendEntries` RPC with a `prevLogIndex` that points beyond the end of your log, you should handle it the same as if you did have that entry but the term did not match (i.e., reply false).
- Check 2 for the `AppendEntries` RPC handler should be executed even if the leader didn't send any entries.
- The `min` in the final step (#5) of `AppendEntries` is necessary, and it needs to be computed with the index of the last new entry. It is not sufficient to simply have the function that applies things from your log between `lastApplied` and `commitIndex` stop when it reaches the end of your log. This is because you may have entries in your log that differ from the leader's log after the entries that the leader sent you (which all match the ones in your log). Because #3 dictates that you only truncate your log if you have conflicting entries, those won't be removed, and if `leaderCommit` is beyond the entries the leader sent you, you may apply incorrect entries.
- It is important to implement the "up-to-date log" check exactly as described in section 5.4. No cheating and just checking the length!

**Failure to follow The Rules**

While the Raft paper is very explicit about how to implement each RPC handler, it also leaves the implementation of a number of rules and invariants unspecified. These are listed in the "Rules for Servers" block on the right hand side of Figure 2. While some of them are fairly self-explanatory, the are also some that require designing your application very carefully so that it does not violate The Rules:

- If `commitIndex > lastApplied` at any point during execution, you should apply a particular log entry. It is not crucial that you do it straight away (for example, in the `AppendEntries` RPC handler), but it is important that you ensure that this application is only done by one entity. Specifically, you will need to either have a dedicated "applier", or to lock around these applies, so that some other routine doesn't also detect that entries need to be applied and also tries to apply.
- Make sure that you check for `commitIndex > lastApplied` either periodically, or after `commitIndex` is updated (i.e., after `matchIndex` is updated). For example, if you check `commitIndex` at the same time as sending out `AppendEntries` to peers, you may have to wait until the next entry is appended to the log before applying the entry you just sent out and got acknowledged.
- If a leader sends out an `AppendEntries` RPC, and it is rejected, but not because of log inconsistency (this can only happen if our term has passed), then you should immediately step down, and not update `nextIndex`. If you do, you could race with the resetting of `nextIndex` if you are re-elected immediately.
- A leader is not allowed to update `commitIndex` to somewhere in a previous term (or, for that matter, a future term). Thus, as the rule says, you specifically need to check that `log[N].term == currentTerm`. This is because Raft leaders cannot be sure an entry is actually committed (and will not ever be changed in the future) if it's not from their current term. This is illustrated by Figure 8 in the paper.

One common source of confusion is the difference between `nextIndex` and `matchIndex`. In particular, you may observe that `matchIndex = nextIndex - 1`, and simply not implement `matchIndex`. This is not safe. While `nextIndex` and `matchIndex` are generally updated at the same time to a similar value (specifically, `nextIndex = matchIndex + 1`), the two serve quite different purposes. `nextIndex` is a guess as to what prefix the leader shares with a given follower. It is generally quite optimistic (we share everything), and is moved backwards only on negative responses. For example, when a leader has just been elected, `nextIndex` is set to be index index at the end of the log. In a way, `nextIndex` is used for performance – you only need to send these things to this peer.

`matchIndex` is used for safety. It is a conservative measurement of what prefix of the log the leader shares with a given follower. `matchIndex` cannot ever be set to a value that is too high, as this may cause the `commitIndex` to be moved too far forward. This is why `matchIndex` is initialized to -1 (i.e., we agree on no prefix), and only updated when a follower positively acknowledges an `AppendEntries` RPC.

**Term confusion**

Term confusion refers to servers getting confused by RPCs that come from old terms. In general, this is not a problem when receiving an RPC, since the rules in Figure 2 say exactly what

you should do when you see an old term. However, Figure 2 generally doesn't discuss what you should do when you get old RPC replies. From experience, we have found that by far the simplest thing to do is to first record the term in the reply (it may be higher than your current term), and then to compare the current term with the term you sent in your original RPC. If the two are different, drop the reply and return. Only if the two terms are the same should you continue processing the reply. There may be further optimizations you can do here with some clever protocol reasoning, but this approach seems to work well. And not doing it leads down a long, winding path of blood, sweat, tears and despair.

A related, but not identical problem is that of assuming that your state has not changed between when you sent the RPC, and when you received the reply. A good example of this is setting `matchIndex = nextIndex - 1`, or `matchIndex = len(log)` when you receive a response to an RPC. This is not safe, because both of those values could have been updated since when you sent the RPC. Instead, the correct thing to do is update `matchIndex` to be `prevLogIndex + len(entries[])` from the arguments you sent in the RPC originally.

**An aside on optimizations**

The Raft paper includes a couple of optional features of interest. In 6.824, we require the students to implement two of them: log compaction (section 7) and accelerated log backtracking (top left hand side of page 8). The former is necessary to avoid the log growing without bound, and the latter is useful for bringing stale followers up to date quickly.

These features are not a part of "core Raft", and so do not receive as much attention in the paper as the main consensus protocol. Log compaction is covered fairly thoroughly (in Figure 13), but leaves out some design details that you might miss if you read it too casually:

- When snapshotting application state, you need to make sure that the application state corresponds to the state following some known index in the Raft log. This means that the application either needs to communicate to Raft what index the snapshot corresponds to, or that Raft needs to delay applying additional log entries until the snapshot has been completed.

- The text does not discuss the recovery protocol for when a server crashes and comes back up now that snapshots are involved. In particular, if Raft state and snapshots are committed separately, a server could crash between persisting a snapshot and persisting the updated Raft state. This is a problem, because step 7 in Figure 13 dictates that the Raft log covered by the snapshot must be discarded.

  If, when the server comes back up, it reads the updated snapshot, but the outdated log, it may end up applying some log entries that are already contained within the snapshot. This happens since the `commitIndex` and `lastApplied` are not persisted, and so Raft doesn't know that those log entries have already been applied. The fix for this is to in-

troduce a piece of persistent state to Raft that records what "real" index the first entry in Raft's persisted log corresponds to. This can then be compared to the loaded snapshot's `lastIncludedIndex` to determine what elements at the head of the log to discard.

The accelerated log backtracking optimization is very underspecified, probably because the authors do not see it as being necessary for most deployments. It is not clear from the text exactly how the conflicting index and term sent back from the client should be used by the leader to determine what `nextIndex` to use. We believe the protocol the authors probably want you to follow is:

- If a follower does not have `prevLogIndex` in its log, it should return with `conflictIndex = len(log)` and `conflictTerm = None`.
- If a follower does have `prevLogIndex` in its log, but the term does not match, it should return `conflictTerm = log[prevLogIndex].Term`, and then search its log for the first index whose entry has term equal to `conflictTerm`.
- Upon receiving a conflict response, the leader should first search its log for `conflictTerm`. If it finds an entry in its log with that term, it should set `nextIndex` to be the one beyond the index of the last entry in that term in its log.
- If it does not find an entry with that term, it should set `nextIndex = conflictIndex`.

A half-way solution is to just use `conflictIndex` (and ignore `conflictTerm`), which simplifies the implementation, but then the leader will sometimes end up sending more log entries to the follower than is strictly necessary to bring them up to date.

## Applications on top of Raft

When building a service on top of Raft (such as the key/value store in the second 6.824 Raft lab, the interaction between the service and the Raft log can be tricky to get right. This section details some aspects of the development process that you may find useful when building your application.

### Applying client operations

You may be confused about how you would even implement an application in terms of a replicated log. You might start off by having your service, whenever it receives a client request, send that request to the leader, wait for Raft to apply something, do the operation the client asked for, and then return to the client. While this would be fine in a single-client system, it does not work for concurrent clients.

Instead, the service should be constructed as a state machine where client operations transition the machine from one state to another. You should have a loop somewhere that takes one client operation at the time (in the same order on all servers – this is where Raft comes

in), and applies each one to the state machine in order. This loop should be the only part of your code that touches the application state (the key/value mapping in 6.824). This means that your client-facing RPC methods should simply submit the client's operation to Raft, and then wait for that operation to be applied by this "applier loop". Only when the client's command comes up should it be executed, and any return values read out. Note that this includes read requests!

This brings up another question: how do you know when a client operation has completed? In the case of no failures, this is simple – you just wait for the thing you put into the log to come back out (i.e., be passed to `apply()`). When that happens, you return the result to the client. However, what happens if there are failures? For example, you may have been the leader when the client initially contacted you, but someone else has since been elected, and the client request you put in the log has been discarded. Clearly you need to have the client try again, but how do you know when to tell them about the error?

One simple way to solve this problem is to record where in the Raft log the client's operation appears when you insert it. Once the operation at that index is sent to `apply()`, you can tell whether or not the client's operation succeeded based on whether the operation that came up for that index is in fact the one you put there. If it isn't, a failure has happened and an error can be returned to the client.

## Duplicate detection

As soon as you have clients retry operations in the face of errors, you need some kind of duplicate detection scheme – if a client sends an `APPEND` to your server, doesn't hear back, and re-sends it to the next server, your `apply()` function needs to ensure that the `APPEND` isn't executed twice. To do so, you need some kind of unique identifier for each client request, so that you can recognize if you have seen, and more importantly, applied, a particular operation in the past. Furthermore, this state needs to be a part of your state machine so that all your Raft servers eliminate the same duplicates.

There are many ways of assigning such identifiers. One simple and fairly efficient one is to give each client a unique identifier, and then have them tag each request with a monotonically increasing sequence number. If a client re-sends a request, it re-uses the same sequence number. Your server keeps track of the latest sequence number it has seen for each client, and simply ignores any operation that it has already seen.

## Hairy corner-cases

If your implementation follows the general outline given above, there are at least two subtle issues you are likely to run into that may be hard to identify without some serious debugging. To save you some time, here they are:

**Re-appearing indices**: Say that your Raft library has some method `Start()` that takes a command, and return the index at which that command was placed in the log (so that you know when to return to the client, as discussed above). You might assume that you will never see `Start()` return the same index twice, or at the very least, that if you see the same index again, the command that first returned that index must have failed. It turns out that neither of these things are true, even if no servers crash.

Consider the following scenario with five servers, S1 through S5. Initially, S1 is the leader, and its log is empty.

1. Two client operations (C1 and C2) arrive on S1
2. `Start()` return 1 for C1, and 2 for C2.
3. S1 sends out an `AppendEntries` to S2 containing C1 and C2, but all its other messages are lost.
4. S3 steps forward as a candidate.
5. S1 and S2 won't vote for S3, but S3, S4, and S5 all will, so S3 becomes the leader.
6. Another client request, C3 comes in to S3.
7. S3 calls `Start()` (which returns 1)
8. S3 sends an `AppendEntries` to S1, who discards C1 and C2 from its log, and adds C3.
9. S3 fails before sending `AppendEntries` to any other servers.
10. S1 steps forward, and because its log is up-to-date, it is elected leader.
11. Another client request, C4, arrives at S1
12. S1 calls `Start()`, which returns 2 (which was also returned for `Start(C2)`.
13. All of S1's `AppendEntries` are dropped, and S2 steps forward.
14. S1 and S3 won't vote for S2, but S2, S4, and S5 all will, so S2 becomes leader.
15. A client request C5 comes in to S2
16. S2 calls `Start()`, which returns 3.
17. S2 successfully sends `AppendEntries` to all the servers, which S2 reports back to the servers by including an updated `leaderCommit = 3` in the next heartbeat.

Since S2's log is `[C1 C2 C5]`, this means that the entry that committed (and was applied at all servers, including S1) at index 2 is C2. This despite the fact that C4 was the last client operation to have returned index 2 at S1.

**The four-way deadlock**: All credit for finding this goes to Steven Allen, another 6.824 TA. He found the following nasty four-way deadlock that you can easily get into when building applications on top of Raft.

Your Raft code, however it is structured, likely has a `Start()`-like function that allows the application to add new commands to the Raft log. It also likely has a loop that, when `commitIndex` is updated, calls `apply()` on the application for every element in the log between `lastApplied` and `commitIndex`. These routines probably both take some lock `a`. In your Raft-based application,

you probably call Raft's `Start()` function somewhere in your RPC handlers, and you have some code somewhere else that is informed whenever Raft applies a new log entry. Since these two need to communicate (i.e., the RPC method needs to know when the operation it put into the log completes), they both probably take some lock `b`.

In Go, these four code segments probably look something like this:

```
func (a *App) RPC(args interface{}, reply interface{}) {
    // ...
    a.mutex.Lock()
    i := a.raft.Start(args)
    // update some data structure so that apply knows to poke us later
    a.mutex.Unlock()
    // wait for apply to poke us
    return
}
```

```
func (r *Raft) Start(cmd interface{}) int {
    r.mutex.Lock()
    // do things to start agreement on this new command
    // store index in the log where cmd was placed
    r.mutex.Unlock()
    return index
}
```

```
func (a *App) apply(index int, cmd interface{}) {
    a.mutex.Lock()
    switch cmd := cmd.(type) {
    case GetArgs:
        // do the get
        // see who was listening for this index
        // poke them all with the result of the operation
    // ...
    }
    a.mutex.Unlock()
}
```

```
func (r *Raft) AppendEntries(...) {
    // ...
    r.mutex.Lock()
    // ...
    for r.lastApplied < r.commitIndex {
      r.lastApplied++
      r.app.apply(r.lastApplied, r.log[r.lastApplied])
    }
    // ...
    r.mutex.Unlock()
}
```

Consider now if the system is in the following state:

- `App.RPC` has just taken `a.mutex` and called `Raft.Start`
- `Raft.Start` is waiting for `r.mutex`
- `Raft.AppendEntries` is holding `r.mutex`, and has just called `App.apply`

We now have a deadlock, because:

- `Raft.AppendEntries` won't release the lock until `App.apply` returns.
- `App.apply` can't return until it gets `a.mutex`.
- `a.mutex` won't be released until `App.RPC` returns.
- `App.RPC` won't return until `Raft.Start` returns.
- `Raft.Start` can't return until it gets `r.mutex`.
- `Raft.Start` has to wait for `Raft.AppendEntries`.

There are a couple of ways you can get around this problem. The easiest one is to take `a.mutex` after calling `a.raft.Start` in `App.RPC`. However, this means that `App.apply` may be called for the operation that `App.RPC` just called `Raft.Start` on before `App.RPC` has a chance to record the fact that it wishes to be notified. Another scheme that may yield a neater design is to have a single, dedicated thread calling `r.app.apply` from `Raft`. This thread could be notified every time `commitIndex` is updated, and would then not need to hold a lock in order to apply, breaking the deadlock.

*( revision history )*